Cleared for unlimited release: CL#08-3913

# Flight Software Complexity

Sponsor: NASA OCE Technical Excellence Program

| | |
|---|---|
| JPL Task Lead: | Dan Dvorak |
| GSFC POC: | Lou Hallock |
| JSC POC: | Pedro Martinez, Brian Butcher |
| MSFC POC: | Cathy White, Helen Housch |
| APL POC: | Steve Williams |
| HQ Sponsor: | Adam West |
| Advisors: | John Kelly, Tim Crumbley |

**JPL Team:**

Kevin Barltrop
Dan Dvorak
Gerard Holzmann
Cin-Young Lee
Bob Rasmussen
Kirk Reinholtz
Bob Vargo

# Flight Software Complexity

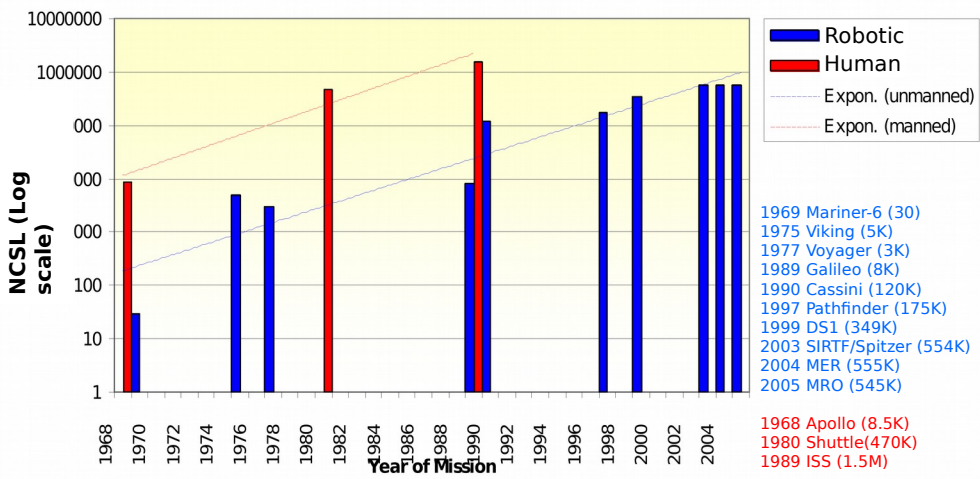
## Charter

Bring forward deployable technical and managerial strategies to effectively address risks from growth in size and complexity of flight software

## Areas of Interest

1. Clear exposé of growth in NASA FSW size and complexity
2. Ways to reduce/manage complexity in general
3. Ways to reduce/manage complexity of fault protection systems
4. Methods of testing complex logic for safety and fault protection provisions

Growth in Code Size for Robotic and Human Missions




## Initiators / Reviewers

- Ken Ledbetter, SMD Chief Engineer
- Stan Fishkind, SOMD Chief Engineer
- Frank Bauer, ESMD Chief Engineer
- George Xenofos, ESMD Dep. Chief Engr.

## Points of Contact

- JPL Dan Dvorak, lead
- GSFC Lou Hallock
- JSC Pedro Martinez
- MSFC Leann Thomas
- APL Steve Williams

# Subtasks and Center Involvement

| | Subtask Description | JPL | GSFC | JSC | MSFC | APL |
|---|---|---|---|---|---|---|
| 1 | **Exposé of growth in NASA flight software** (SI-1) | x | x | x | x | x |
| 2 | **Architectures, trades, and avionics impacts** (SI-2, A1) | x | x | x | x | x |
| 3 | **Literature survey of strategies to manage complexity** (SI-2, A2) | x | | | | |
| 4 | **Position paper on out-of-the-box approach** (SI-2, A4) | x | | | | |
| 5 | **Fault protection workshop (joint with Fault Mgmt Workshop coordinated by MSFC)** (SI-3, A1) | x | x | x | x | x |
| 6 | **Document fault protection used to date within NASA** (SI-3, A2) | x | x | x | x | x |
| 7 | **Integrating fault protection into “nominal” system** (SI-3, A3) | x | | | | |
| 8 | **Testing of complex logic for safety and fault protection** (SI-4) | x | | | | |

# Growth in Flight Software Size

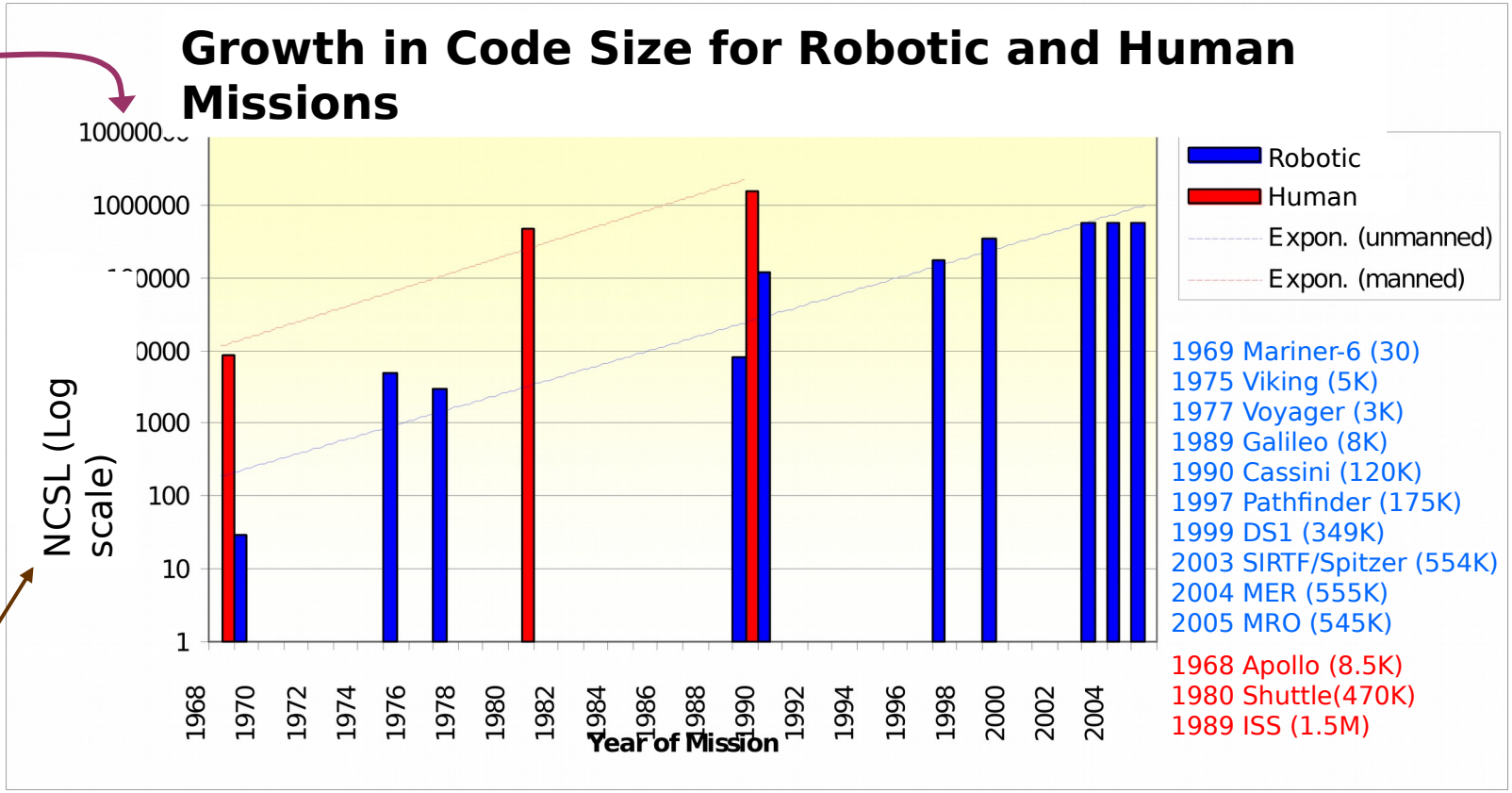
***JPL, JSC:*** ***Significant growth***

***GSFC:*** ***LISA much more complex***

***MSFC, APL:*** ***No growth trend***

# Growth Trends in NASA Flight Software

Note log scale!




NCSL = Non-Comment Source Lines

The 'year' used in this plot is for a mission is typically the year of launch, or of completion of the primary software.
Line counts are either from best available source or direct line counts (e.g., for the JPL and LMA missions).
The line count for Shuttle Software is from Michael King, Space Flight Operations Contract Software Process Owner, April 2005

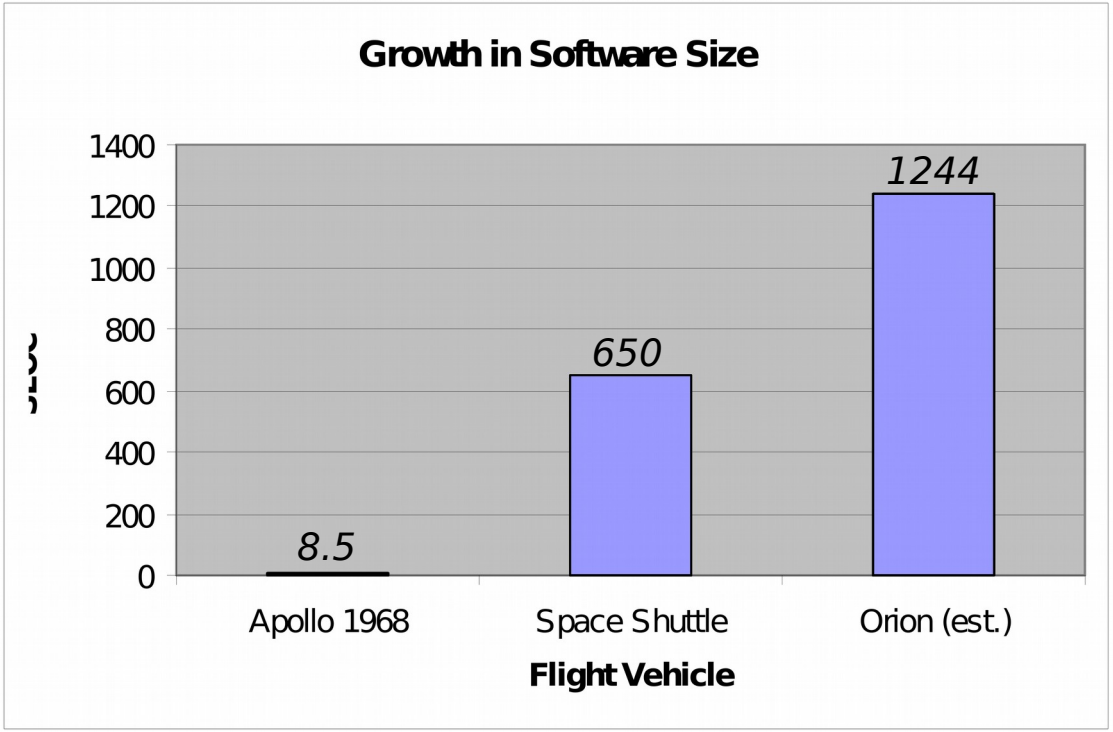
Growth rate ~10X every 10 years

MSL size estimate 1.7M NCSL

Source: Gerard Holzmann, JPL

# Software Growth in Human Spaceflight

JSC data




The Orion (CEV) numbers are current estimates.

To make Space Shuttle and Orion comparable, neither one includes backup flight software since that figure for Orion is TBD.

Source: Pedro Martinez, JSC

# How Big is a Million Lines of Code?

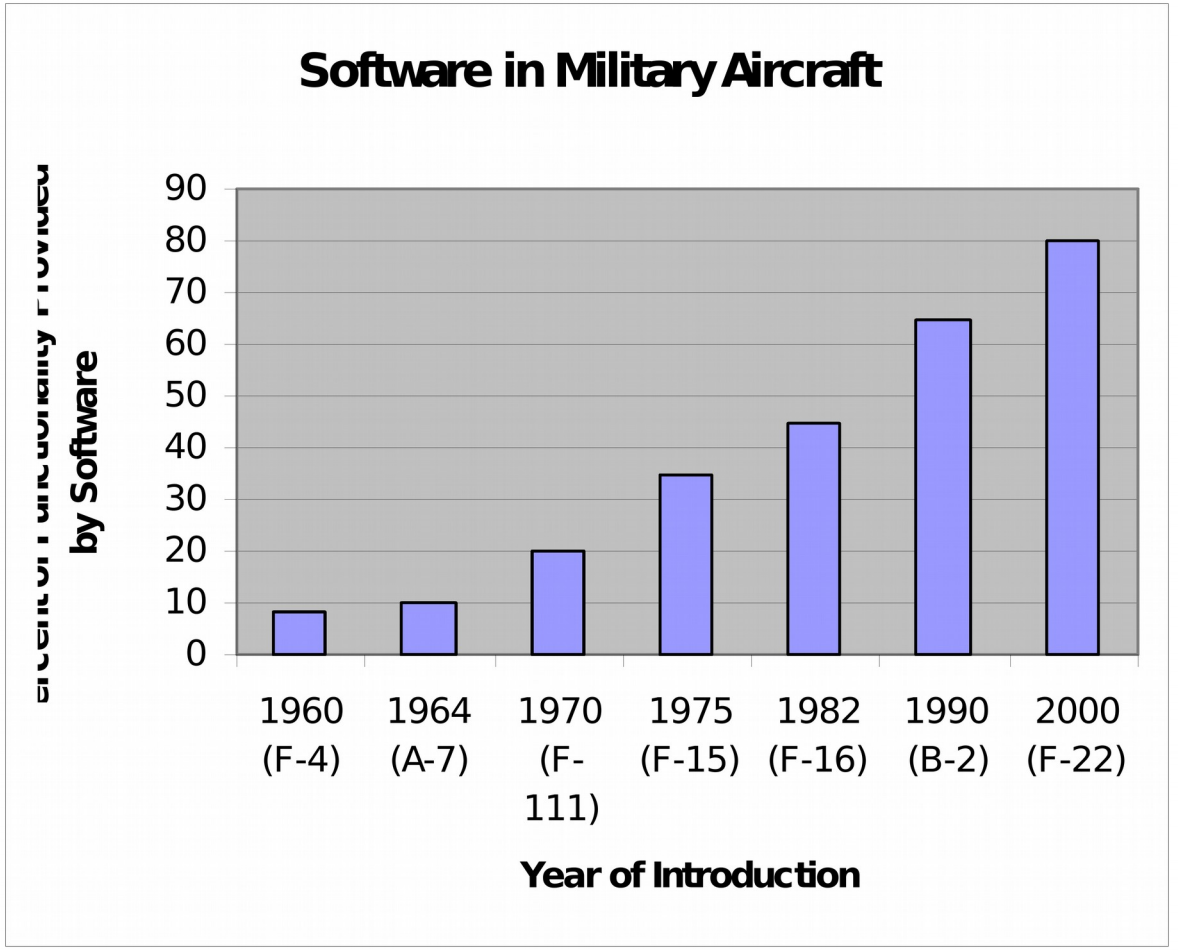
A novel has ~500K characters
00K words × ~5 characters/word)

million-line program has ~20M characters
M lines × ~20 characters/line), or about 40 novels

Source:
Les Hatton,
University of Kent,
*Encyclopedia of Software Engineering*, John Marciniak, editor in

# Software Growth in Military Aircraft

- Flight software is growing because it is providing an increasing percentage of system functionality
- With the newest F-22 in 2000, software controls 80% of everything the pilot does
- Designers put functionality in software or firmware because it is easier and/or cheaper than hardware



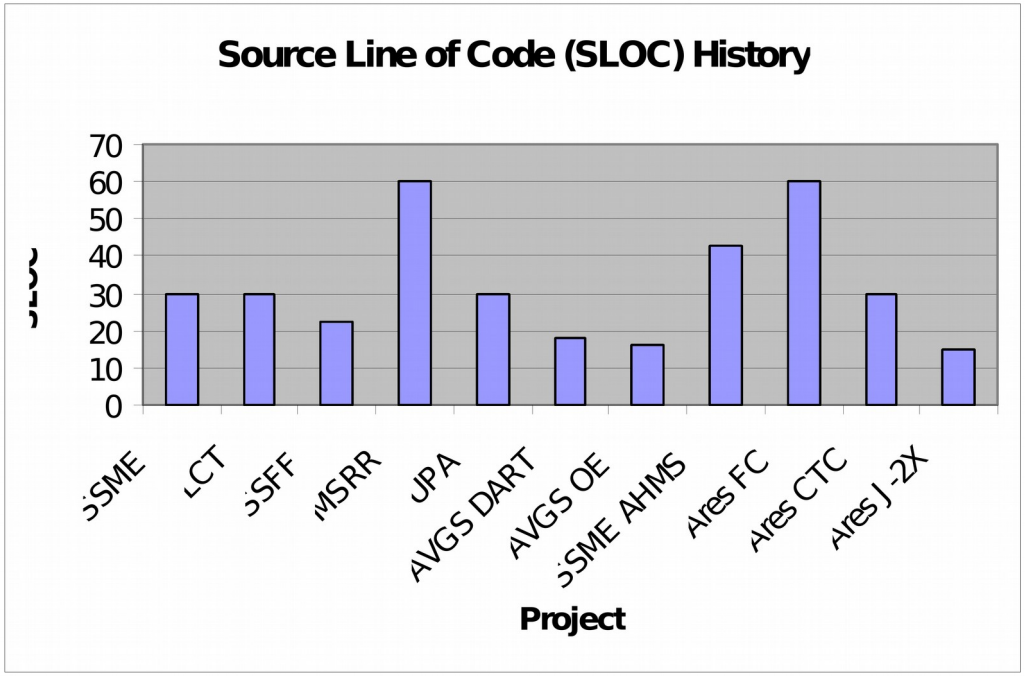

"Crouching Dragon, Hidden Software: Software in DoD Weapon Systems", Jack Ferguson, IEEE Software, vol. 18, no. 4, pp.105-107, Jul/Aug, 2001.

# Size Comparisons of Embedded Software

| System | Lines of Code | Language |
|---|---|---|
| Mars Science Laboratory | 1.7M (est.) | C |
| F-22 Raptor | 2.5M | Ada (90%) |
| Seawolf Submarine Combat System AN/BSY-2 | 3.6M | Ada |
| Boeing 777 | 4M | Ada |
| F-35 Joint Strike Fighter | 5.6M | C and C++ |
| Boeing 787 | 7M | Ada (largely) |
| Typical GM car in 2010 | 100M | MISRA-C for critical systems |

# MSFC Flight Software Organization (no trend)

MSFC data




**SSME** - Space Shuttle Main Engine ~30K SLOC C/assembly (1980's – 2007)

**LCT** - Low Cost Technology (FASTRAC engine) ~30K SLOC C/Ada (1990's)

**SSFF** – Space Station Furnace Facility ~22K SLOC C (cancelled 1997)

**MSRR** – Microgravity Science Research Rack ~60K SLOC C (2001 - 2007)

**UPA** – Urine Processor Assembly ~30K SKOC C (2001 - 2007)

**AVGS DART** – Advanced Video Guidance System for Demonstration of Automated Rendezvous Technology ~18K SLOC C (2002 - 2004)

**AVGS OE** – AVGS for Orbital Express ~16 K SLOC C (2004 - 2006)

**SSME AHMS** – Space Shuttle Main Engine Advanced Health Management System ~42.5K SLOC C/assembly (2006 flight)

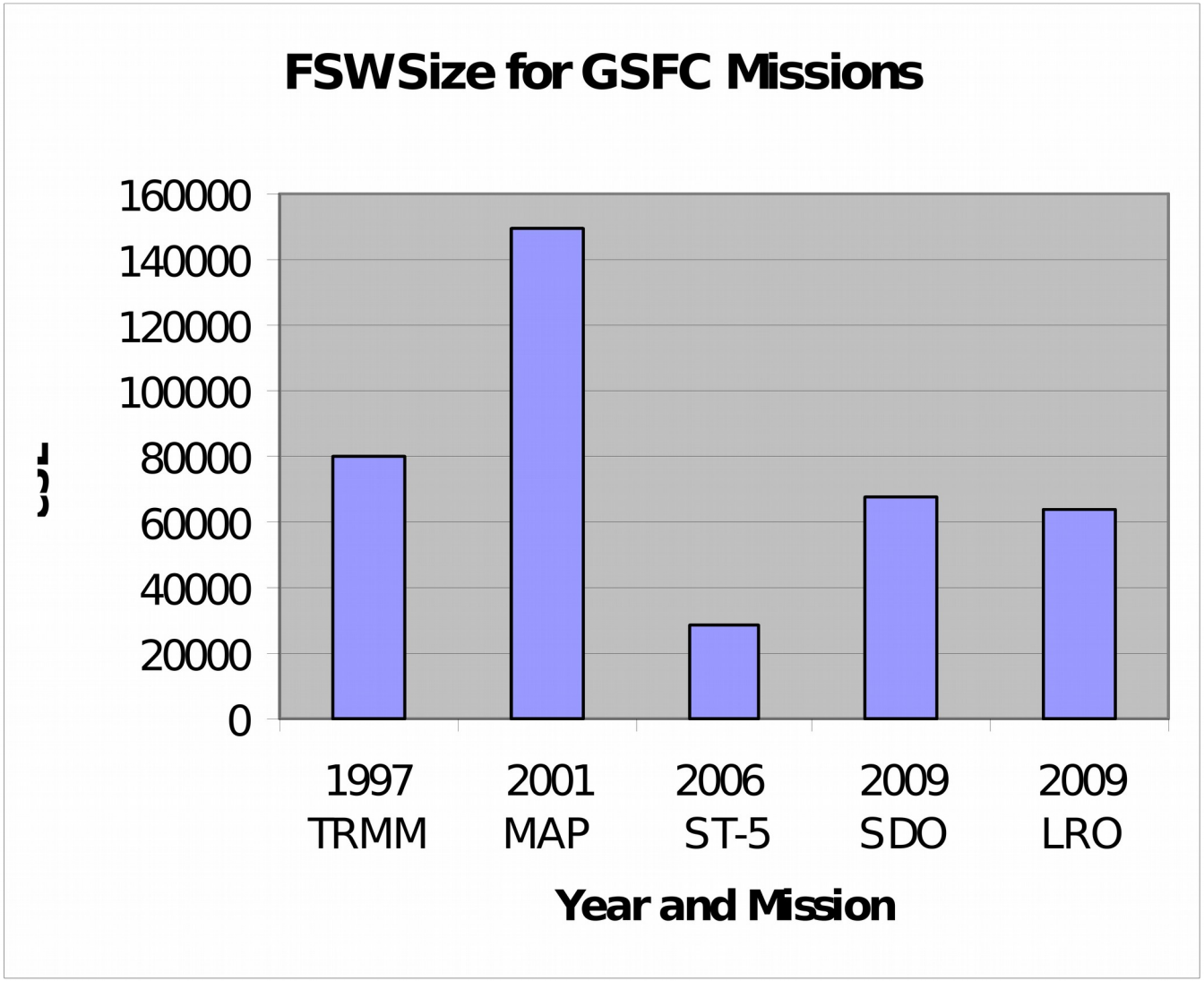
**FC** - Ares Flight Computer estimated ~60K SLOC TBD language (2007 SRR)

**CTC** - Ares Command and Telemetry Computer estimated ~30K SLOC TBD language (2007 SRR)

**Ares J-2X** engine initial estimate ~15K SLOC TBD language (2007 SRR)

Source: Cathy White, MSFC

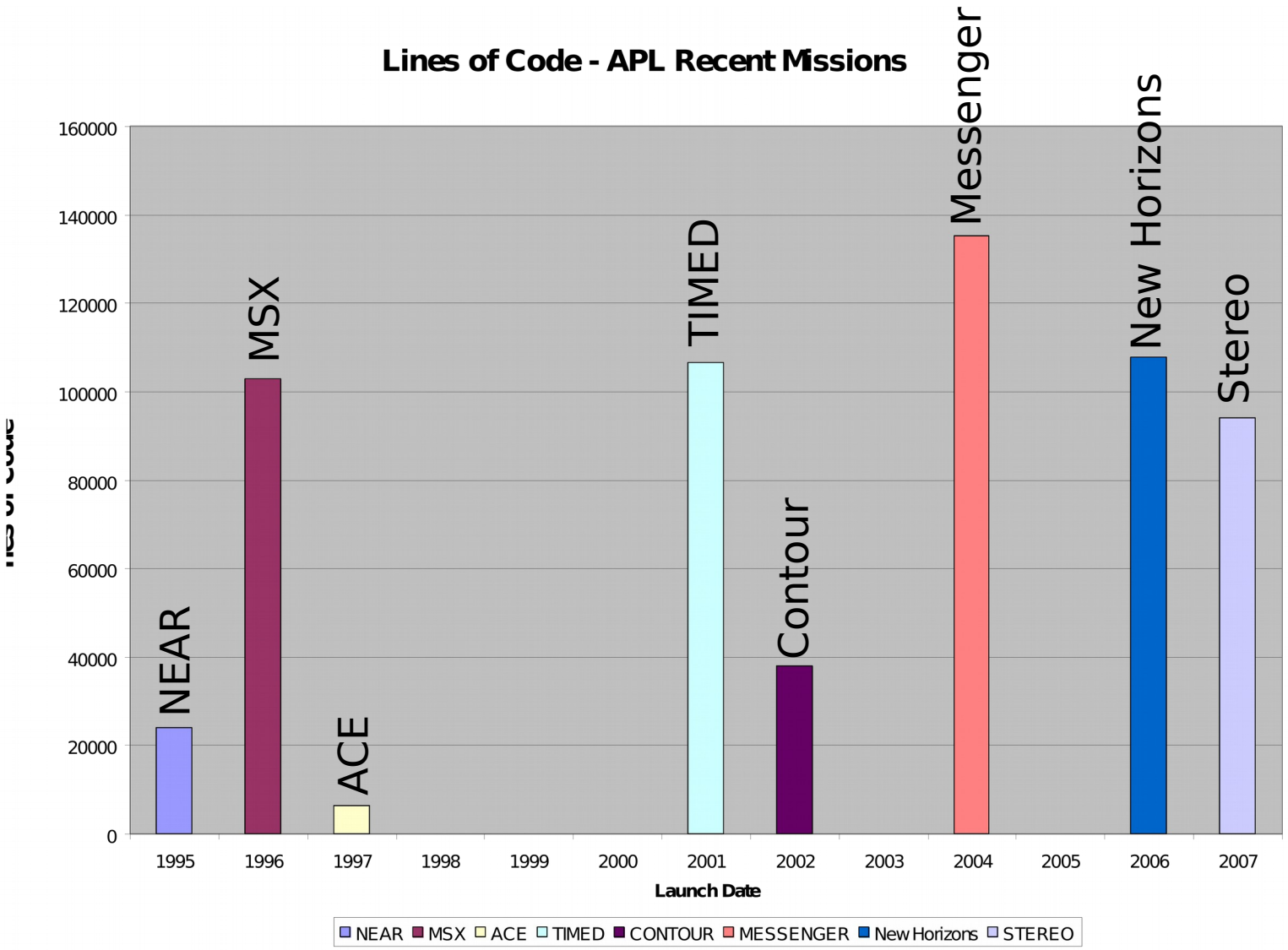
# GSFC Flight Software Sizes (no trend)




Source: David McComas, GSFC

Note: LISA expected to be much larger

# APL Flight Software Sizes (no trend)




Source: Steve Williams, APL

# About Complexity

- ***Software size is a proxy for complexity***
- ***But what <u>is</u> complexity?***
- ***Where does it appear?***
- ***Why is it getting bigger?***

# What is Complexity?

- **Complexity is a measure of how hard something is to understand or achieve**
  - **Components** — How many kinds of things are there to be aware of?
  - **Connections** — How many relationships are there to track?
  - **Patterns** — Can the design be understood in terms of well-defined patterns?
  - **Requirements** — Timing, precision, algorithms

- **Two kinds of complexity:**
  - **Essential Complexity** – How complex is the underlying problem?
  - **Incidental Complexity** – What extraneous complexity have we added?

- **Complexity appears in at least four key areas:**
  - Complexity in requirements
  - Complexity of the software itself
  - Complexity of *testing* the system
  - Complexity of *operating* the system

"Complexity is a total system issue, not just a software issue."

*– Orlando Figueroa*

# Why is Flight Software Growing?

"The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them. …

**It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope**, and it is the growth in utilization of software ponents that is largely responsible for high overall complexity of many system gns."

Michael Lyu

## Causes of Software Growth

# Expanding Functions

**Command sequencing**
**Telemetry collection & formatting**
**Attitude and velocity control**
**Aperture & array pointing**
**Configuration management**
**Payload management**
**Fault detection and diagnosis**
**Safing and fault recovery**
**Critical event sequencing**
- High level commanding
- Profiled pointing and control
- Motion compensation
- Robot arm control
- Data storage management
- Data encoding/decoding
- Data editing and compression
- Parachute deployment
- Guided descent and landing
- Trajectory and ephemeris propagation
- Thermal control
- Star identification
- Feature recognition and target tracking
- Trajectory determination
- Maneuver planning

Source: Bob Rasmussen, JPL

**Momentum management**
**Aerobraking**
**Fine guidance pointing**
**Data priority management**
**Event-driven sequencing**
**Surface sample acquisition and handling**
**Surface mobility and hazard avoidance**
**Relay communications**
**Science event detection**
**Automated planning and scheduling**
**Operation on or near small bodies**

**Guided atmospheric entry**
**Tethered system soft landing**
**Interferometer control**

**Dynamic resource management**
**Long distance traversal**
**Landing hazard avoidance**
**Model-based reasoning**
**Plan repair**
**Guided ascent**
**Rendezvous and docking**
**Formation flying**
**Opportunistic science**
*and so on . . .*

*"Flight software is a system's complexity sponge."*

*Past*

*Planned*

*Future*

***Ever more complicated, and numerous***

# Why LISA is More Complex

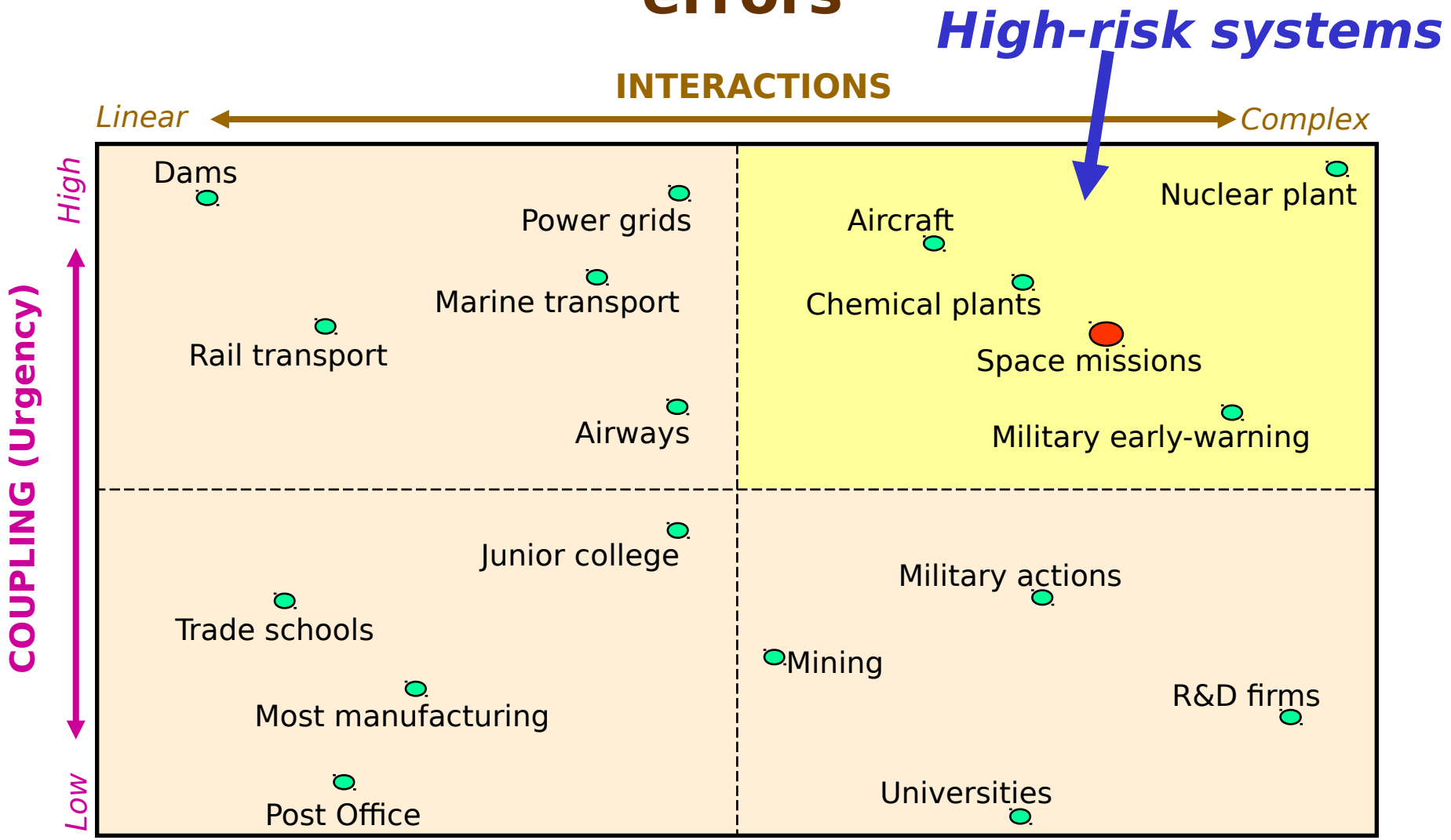
- The Laser Interferometer Space Antenna (LISA) mission represents a significant step-up in FSW complexity
- Spacecraft and payload becomes blurred; the science instrument is created via laser links connecting three spacecraft forming approximately an equilateral triangle of side length 5 million kilometers.

## Sources of Increased Complexity

- The science measurement is formed by measuring to ***extraordinarily high levels of precision*** the distances separating the three spacecraft.
  - Formation flying between a LISA spacecraft and its proof masses must be controlled to within a nanometer or better accuracy.
  - Mispointings on order of milli-arcseconds will disrupt laser links
  - FSW validation will need to see deviations at micro-arcsecond level
- ***Doubling of issues***
  - Twice as many control modes as a typical astrophysics mission
  - Twice as many sensors and actuators
  - Fault detection on twice as many telemetry points
- ***Inputs and outputs larger for control laws and estimators***
- ***New control laws for Drag Free control***

Source: Lou Hallock, GSFC

# raise risk of design defects and operational errors




Source: Charles Perrow, "Normal Accidents: Living with High-Risk Technologies", 1984.

NASA Speech

# Michael Griffin on Complex Systems

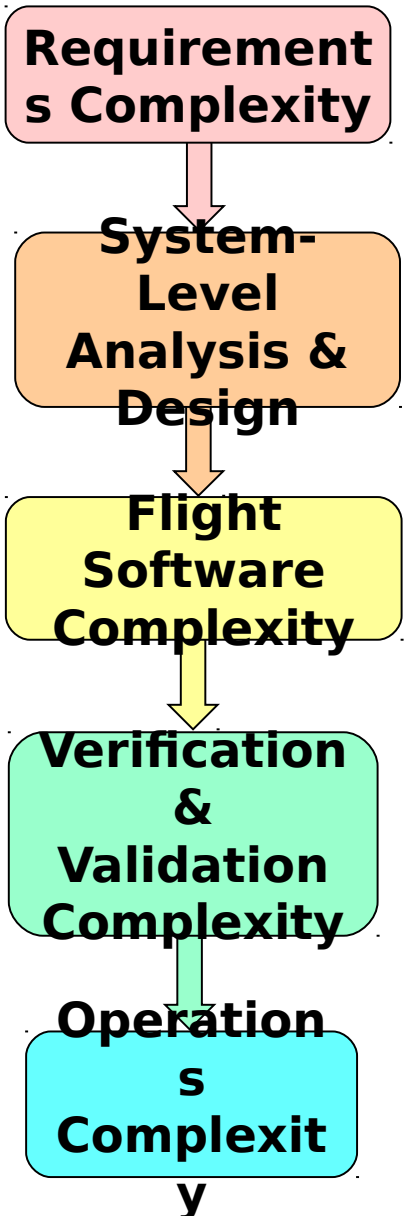
"Complex systems usually come to grief, when they do,
not because they fail to accomplish their nominal purpose. Complex systems typically fail because of the unintended consequences of their design …

"I like to think of system engineering as being fundamentally concerned with minimizing, in a complex artifact, unintended interactions between elements desired to be separate. Essentially, this addresses Perrow's concerns about tightly coupled systems. System engineering seeks to assure that elements of a complex artifact are [illegible]ded."

Michael Griffin, NASA

Substitute "software architecture" for "systems engineering" and it makes equally good sense!

# Scope of Study plus Key Findings




- Challenging requirements raise downstream complexity (unavoidable)
- Lack of requirements rationale permit unnecessary requirements
- Requirements volatility creates a moving target for designers

- Engineering trade studies not done: a missed opportunity
- Architectural thinking/review needed at level of systems and software

- Inadequate software architecture and poor implementation
- General lack of design patterns (and architectural patterns)
- Coding guidelines help reduce defects and improve static analysis
- Descopes often shift complexity to operations

- Growth in testing complexity seen at all centers
- More software components and interactions to test
- COTS software is a mixed blessing

- Shortsighted FSW decisions make operations unnecessarily complex
- Numerous “operational workarounds” raise risk of command errors

# Detailed Recommendations

| | | |
|---|---|---|
| **Complexity Awareness** | | |
| | **R1** Education about downstream effects of decisions | Link |
| **Project Management** | | |
| | **R2** Emphasize requirements rationale | Link |
| | **R3** Serious attention to trade studies | Link |
| | **R7** Involve operations engineers early and often | Link |
| | **R10** Technical kickoff for projects | Link |
| | **R16** Use software metrics | Link |
| **Architecture** | | |
| | **R4** More up-front analysis and architecting | Link |
| | **R5** Architecture review board | Link |
| | **R6** Grow and promote software architects | Link |
| | **R9** Invest in a reference architecture | Link |

*Recommendation 1*

# Education about "effect of *x* on complexity"

- **Finding: Engineers and scientists often don't realize the downstream complexity entailed by their decisions**
  - Seemingly simple science "requirements" and avionics designs can have large impact on *software* complexity, and software decisions can have large impact on *operational* complexity
- **Recommendations:**
  - **Educate engineers about the *kinds* of decisions that affect complexity**
    - Intended for systems engineers, subsystem engineers, instrument designers, scientists, flight and ground software engineers, and operations engineers
  - **Include complexity analysis as part of reviews**
- **Options:**
  1. Create a "Complexity Primer" on a NASA-internal web site (link)
  2. Populate NASA Lessons Learned with complexity lessons
  3. Publish a paper about common causes of complexity

## *Recommendation 2*

# Emphasize Requirements Rationale

- **Finding: Unsubstantiated requirements have caused unnecessary complexity. Rationale for requirements often missing or superficial or misused.**
- **Recommendation: Require rationales at Levels 1, 2, 3**
  - Rationale explains *why* a requirement exists
  - Numerical values require strong justification (e.g. “99% data completeness”, “20 msec response”, etc). Why *that* value rather than an easier value?
- **Note:** NPR 7123, NASA System Engineering Requirements, specifies in an appendix of “best typical practices” that requirements include rationale, but offers no guidance on how to write a good rationale or check it. NASA Systems Engineering Handbook provides some guidance (p. 48).
- **Options:**
  1. Projects should create a “rationale document” for a set of requirements (sometimes better than rationale for individual requirements):
     - What the mission is trying to accomplish
     - What the trade studies showed
     - What needs to be done
  2. Encourage local procedures that mandate rationale
  3. Add to NASA Lessons Learned about lack of rationale
  4. Development team should inform project management about hard-to-meet requirements

## *Recommendation 3*

# Serious Attention to Trade Studies

- **Finding: Engineering trade studies often not done or done superficially or done too late**
  - Kinds of trade studies: flight vs. ground, hardware vs. software vs. firmware (including FPGAs), FSW vs. mission ops and ops tools
  - Possible reasons: schedule pressure, unclear ownership, culture
- **Recommendation: Ensure that trade studies are properly staffed, funded, and done early enough**

"As the line between systems and software engineering blurs, multidisciplinary approaches and teams are becoming imperative."

— Jack Ferguson
Director of Software Intensive Systems, DoD
IEEE Software, July/August 2001

IEEE Software, July/August 2001

## Recommendation 4

# More Up-Front Analysis & Architecting

- **Finding: There are clear trends of increasing complexity in NASA missions**
  - Complexity is evident in requirements, FSW, testing, and ops
  - We can reduce *incidental* complexity through better architecture

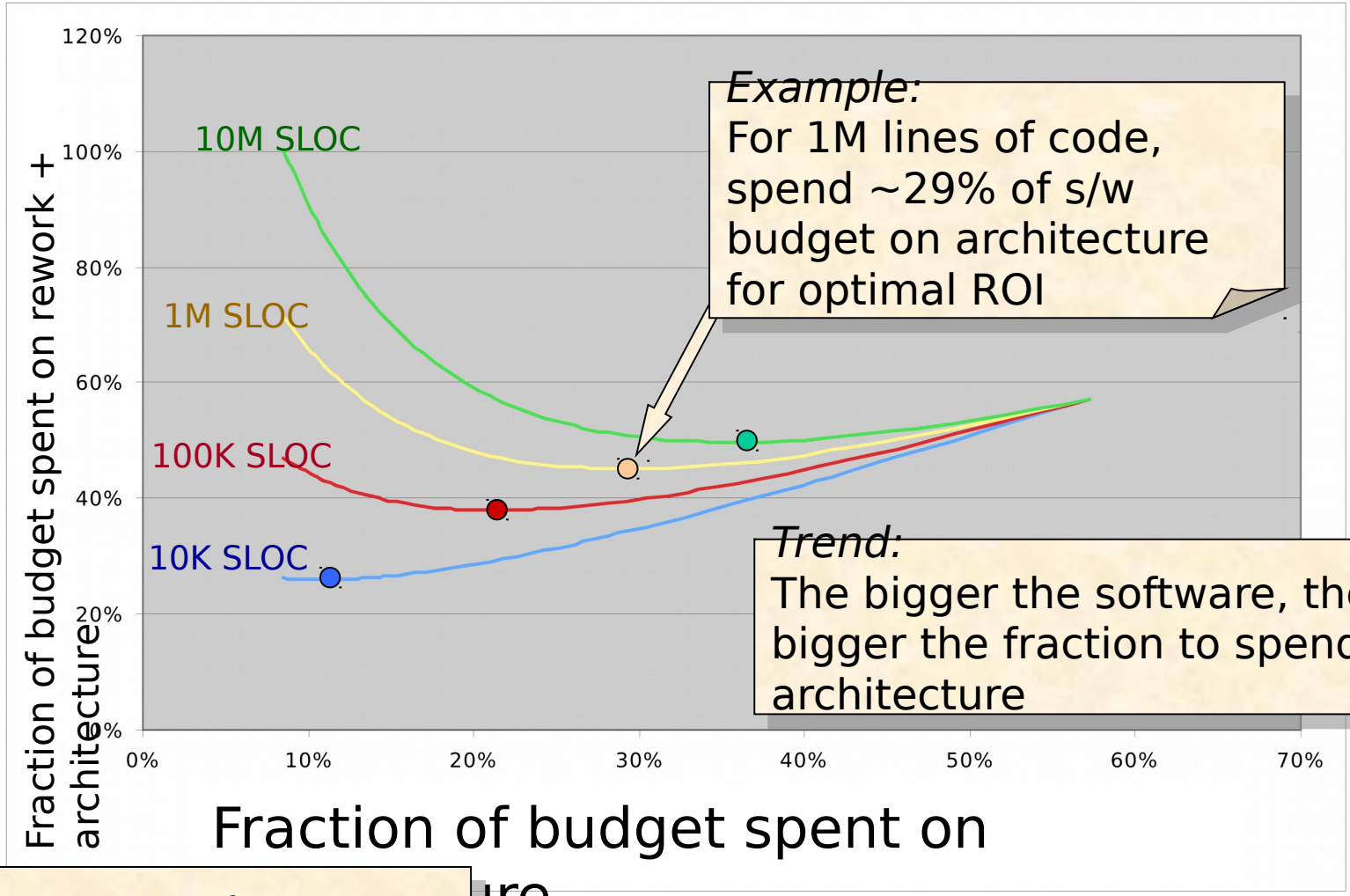
"Point of view is worth 80 IQ points."

– Alan Kay, 1982 (famous computer scientist)

- **Recommendation: Spend more time up front in requirements analysis and architecture to *really* understand the job and its solution** (What is architecture?)
  - Architecture is an essential systems engineering responsibility, and the architecture of behavior largely falls to software
  - Cheaper to deal with complexity early in analysis and architecture
  - Integration & testing becomes easier with well-defined interfaces and well-understood interactions
  - Be aware of Conway's Law (any piece of software reflects the organizational structure that produced it)

# Architecture Investment "Sweet Spot"

Predictions from COCOMO II model for software cost estimation




*Note:*
Prior investment in a reference architecture pays dividends

Source: Kirk Reinholtz, JPL

## Recommendation 5

# Software Architecture Review Board

- **Finding: In the 1990's AT&T had a standing Architecture Review Board that examined proposed software architectures for projects, in depth, and pointed out problem areas for rework**
  - The board members were experts in architecture & system analysis
  - They could spot common problems a mile away
  - The review was *invited* and the board provided *constructive* feedback
  - It helped immensely to avoid big problems

- **Recommendation: Create a *professional* architecture review board and add architecture reviews as a best practice (details)**

Maybe similar to Navigation Advisory Group (NAG)

- **Options:**
  1. Insert architecture gates into existing NASA documents
  2. Leverage existing checklists for architecture reviews [8]
  3. Organize a set of "architectural lessons learned"
  4. Consider reviewers from academia and industry for very large projects

*Recommendation 6*

# Grow and Promote Software Architects

- **Finding: Software architecture is vitally important in reducing incidental complexity, but architecture skills are uncommon and need to be nurtured**
  - Reference: (what is architecture?)  (what is an architect?)
- **Recommendation: Increase the ranks of software architects and put them in positions of authority**
- **Options:**
  1. Target experienced software architects for strategic hiring
  2. Nurture budding architects through education and mentoring (think in terms of a 2-year Master's program)
  3. Expand APPEL course offerings:
     - Help systems engineers to think architecturally
     - The architecture of *behavior* largely falls to software, and systems engineers must understand how to analyze control flow, data flow, resource management, and other cross-cutting issues

*Recommendation 7*

# Involve Operations Engineers Early & Often

- **Findings that increase ops complexity:**
  - Flight/ground trades and subsequent FSW descope decisions often lack operator input
  - Shortsighted decisions about telemetry design, sequencer features, data management, autonomy, and testability
  - Large stack of "operational workarounds" raise risk of command errors and distract operators from vigilant monitoring

Findings are from a "gripe session on ops complexity" held at JPL

- **Recommendations:**
  - Include experienced operators in flight/ground trades and FSW descope decisions
  - Treat operational workarounds as a cost and risk upper; *quantify* their cost
  - Design FSW to allow tests to start at several well-known states (shouldn't have to "launch" spacecraft for each test!)

## *Recommendation 8*

# Analyze COTS for Testing Complexity

**COTS software is a mixed blessing**

- **Finding: COTS software provides valuable functionality, but often comes with numerous other features that are not needed. However, the unneeded features often entail extra testing to check for undesired interactions.**
- **Recommendation: In make/buy decisions, analyze COTS software for separability of its components and features, and thus their effect on testing complexity**
  - Weigh the cost of testing unwanted features against the cost of implementing only the desired features

# Cautionary Note

Some recommendations are common sense, but aren't common practice. Why not? Some reasons below.

❶ **Cost and schedule pressure**
- Some recommendations require time and training, and the benefits are hard to quantify up front

❷ **Lack of Enforcement**
- Some ideas already exist in NASA requirements and local practices, but aren't followed because of ❶ and because nobody checks for them

❸ **Pressure to inherit from previous mission**
- Inheritance can be a very good thing, but "inheritance mentality" inhibits new ideas, tools, and methodologies

❹ **No incentive to "wear the big hat"**
- Project managers focus on point solutions for their missions, with no infrastructure investment for the future

## *Summary*

# Big-Picture Take-Away Message

- **Flight software growth is exponential, and will continue**
  - Driven by ambitious requirements
  - More easily accommodates new functions
  - Naturally accommodates evolving understanding
- **Complexity decreases with …**
  - Substantiated, unambiguous, testable requirements
  - Awareness of downstream effects of engineering decisions
  - Well-chosen architectural patterns, design patterns, and coding guidelines
  - Fault protection integrated into nominal control (not an add-on)
  - Faster processors and larger memories (timing and memory margin)
  - Careful use of COTS software
- **Architecture addresses complexity**
  - Confront complexity at the start (can't test away complexity)
  - Need more architectural thinkers (education, career path)
  - Architecture reviews (follow AT&T's example)
  - See "Thinking Outside the Box" for how to think architecturally

# Epilogue

- **Angst about software complexity in 2008 is the same as in 1968** (See NATO 1968 report, slide)
  - We build systems to the limit of our ability
  - In 1968, 10K lines of code was complex
  - Now, 1M lines of code is complex, for the same price

"While technology can change quickly, getting your people to change takes a great deal longer. That is why the people-intensive job of developing software has had essentially the same problems for over 40 years. It is also why, unless you do something, the situation won't improve by itself. In fact, current trends suggest that your future products will use more software and be more complex than those of today. This means that more of your people will work on software and that their work will be harder to track and more difficult to manage. **Unless you make some changes in the way your software work is done, your current problems will likely get much worse**."

*Winning with Software: An Executive Strategy*, 2001

# Reserve Slides

***Other recommendations***

***Other growth charts***

***Other observations about NASA software***

***Educational notes about software***

# Hyperlinks to Reserve Slides

- R9: Invest in reference arch.
- R10: Technical kickoff
- R11: Use static analysis tools
- R12: Fault protection terminology
- R13: Fault protection review
- R14: Fault protection education
- R15: Fund fault containment
- R16: Use software metrics
- Software metrics concerns
- Fault Management Workshop
- Flight software characteristics
- Two kinds of complexity
- Sources of complexity
- Dietrich Döerner on complexity
- FSW growth trend
- Growth in GM auto s/w
- Residual defects in s/w
- Software development process
- State-of-art testing methods
- Limits to software size?
- Impediments within NASA
- Poor practices in NASA
- NATO 1968 s/w conference
- Source lines of code
- What is Architecture?
- What is an Architect?
- What is s/w architecture?
- What is a reference arch?
- AT&T's architecture reviews
- What is static analysis?
- What is cyclomatic complexity?
- No silver bullet
- Aerospace Corp. activities
- Software complexity primer
- Audiences briefed
- References

# Invest in Reference Architecture & Core Assets

- **Finding: Although each mission is unique, they must all address common problems: attitude control, navigation, data management, fault protection, command handling, telemetry, uplink, downlink, etc. Establishment of uniform patterns for such functionality, across projects, saves time and mission-specific training. This requires investment, but project managers have no incentive to "wear the big hat"**
- **Recommendation: Earmark funds for development of a reference architecture (a predefine architectural pattern) and core assets, at each center, to be led and sustained by the appropriate technical line organization, with senior management support**
  - A reference architecture embodies a huge set of lessons learned, best practices, architectural principles, design patterns, etc.

Key

- **Options:**
  1. Create a separate fund for reference architecture (infrastructure investment)
  2. Keep a list of planned improvements that projects can contribute to, their intended contribution

*See backup slide on reference architecture*

*Recommendation 10*

# Formalize a 'Technical Kickoff' for Projects

- **Finding: Flight project engineers move from project to project, often with little time to catch up on technology advances, so they tend to use the same old stuff**
- **Recommendation:**
  - **Option 1: Hold 'technical kickoff meetings' for projects as a way to infuse new ideas and best practices, and create champions within the project**
    - Inspire rather than mandate
    - Introduces new architectures, processes, tools, and lessons
    - Supports technical growth of engineers
  - **Option 2: Provide 4-month "sabbatical" for project engineers to learn a TRL 6 software technology, experiment with it, give feedback for improvements, and then infuse it**

Michael Aguilar, NESC, is a strong proponent

- **Steps:**
  1. Outline a structure and a technical agenda for a kickoff meeting
  2. Create a well-structured web site with kickoff materials
  3. Pilot a technical kickoff on a selected mission

## *Recommendation 11*

# Static Analysis for Software

- **Finding: Commercial tools for static analysis of source code are mature and effective at detecting many kinds of software defects, but are not widely used**
  - Example tools: Coverity, Klocwork, CodeSonar
  - Michael Aguilar of NESC in strong agreement
- **Recommendation: Provide funds for: (a) site licenses of source code analyzers at flight centers, and (b) local guidance and support**
- **Notes:**
  1. Poll experts within NASA and industry regarding best tools for C, C++, and Java
  2. JPL provides site licenses for Coverity and Klocwork

*Reference*

# What is Static Analysis?

- **Static code analysis** is the analysis of computer software that is performed without actually executing programs built from that software. In most cases analysis is performed on the source code.
- Kinds of problems that static analysis can detect:
  - Null pointer dereference
  - Use after free
  - Double free
  - Dead code due to logic errors
  - Uninitialized variables
  - Erroneous switch cases
  - Deadlocks
  - Lock contentions
  - Race conditions
  - Memory leaks
  - File handle leaks
  - Database connection leaks
  - Mismatched array new/delete
  - Missing destructor
  - STL usage errors
  - API error handling
  - API ordering checks
  - Array and buffer overrun

Source: *"Controlling Software Complexity: The Business Case for Static Source Code Analysis"*, Coverity, www.coverity.com

Recommendation 12

# Fault Protection Reference Standardization

- **Finding: Inconsistency in the terminology for fault protection among NASA centers and their contractors, and a lack of reference material for which to assess the suitability of fault protection approaches to mission objectives.**
  - Example Terminology: Fault, Failure, Fault Protection, Fault Tolerance, Monitor, Response.
- **Recommendation: Publish a NASA Fault Protection Handbook or Standards Document that provides:**
  - An approved lexicon for fault protection.
  - A set of principles and features that characterize software architectures used for fault protection.
  - For existing and past software architectures, a catalog of recurring design patterns with assessments of their relevance and adherence to the identified principles and features.

*Findings from NASA Planetary Spacecraft Fault Management Workshop*

Source: Kevin Barltrop, JPL

## Recommendation 13

# Fault Protection Proposal Review

- **Finding: The proposal review process does not assess in a consistent manner the risk entailed by a mismatch between mission requirements and the proposed fault protection approach.**
- **Recommendation: For each mission proposal generate an explicit assessment of the match between mission scope and fault protection architecture. Penalize proposals or require follow-up for cases where proposed architecture would be insufficient to support fault coverage scope.**
  - Example: Dawn recognized the fault coverage scope problem, but did not appreciate the difficult of expanding fault coverage using the existing architecture.
  - The handbook or standards document can be used as a reference to aid in the assessment and provide some consistency.

*Findings from NASA Planetary Spacecraft Fault Management Workshop*

Source: Kevin Barltrop, JPL

## Recommendation 14

# Fault Protection Education

- **Finding: Fault protection and autonomy receives little attention within university curricula, especially within engineering programs. This hinders the development of a consistent fault protection culture needed to foster the ready exchange of ideas.**

- **Recommendation: Sponsor or facilitate the addition of a fault protection and autonomy course within a university program, such as a Controls program.**
  - **Example: University of Michigan could add a "Fault Protection and Autonomy Course."**

*Findings from NASA Planetary Spacecraft Fault Management Workshop*

Source: Kevin Barltrop, JPL

## *Recommendation 15*

# Fund R&D on Fault Containment

- **Finding: Given growth trends in flight software, and given current achievable defect rates, the odds of a mission-ending failure are increasing** (see slide 43)
  - A mission with 1 Million lines of flight code, with a low residual defect ratio of 1 per 1000 lines of code, then translates into 900 benign defects, 90 medium, and 9 potentially fatal residual software defects (i.e., these are defects that *will* happen, not those that could happen)
  - Bottom line: As more functionality is done in software, the probability of mission-ending software defects increases (until we get smarter)
- **Recommendation: Extend the concept of onboard fault protection to cover software failures. Develop and test techniques to detect software faults at run-time and contain their effects**
  - One technique: upon fault detection, fall back to a simpler-but-more-verifiable version of the failed software module

## Recommendation 16

# Apply Software Metrics

- **Finding: No consistency in flight software metrics**
  - No consistency in how to measure and categorize software size
  - Hard to assess amount and areas of FSW growth, even *within* a center
  - NPR 7150.2 Section 5.3.1 (Software Metrics Report) requires measures of software progress, functionality, quality, and requirements volatility
- **Recommendations: Development organizations should …**
  - collect software metrics per NPR 7150.2
  - use metrics as a mgmt tool to assess cost, technical, and schedule progress
  - compare to historical data for planning and monitoring
  - seek measures of complexity at code level and architecture level
  - Save flight software from each mission in a repository for undefined future analyses (software archeology)
- **Non-Recommendation: Don't attempt NASA-wide metrics. Better to drive local center efforts.** (See slide)

"The 777 marks the first time The Boeing Company has applied software metrics uniformly across a new commercial-airplane programme. This was done to ensure simple, consistent communication of information pertinent to software schedules among Boeing, its software suppliers, and its customers—at all engineering and management levels. In the short term, uniform application of software metrics has resulted in improved visibility and reduced risk for 777 on-board software."

Robert Lytz, "Software metrics for the Boeing 777: a case study", Software Quality Journal, Springer Netherlands

# Difficulties of Software Metrics

## An earlier attempt to define NASA-wide software metrics foundered on issues such as these

### Concerns

- Will the data be used to:
  - compare productivity among centers?
  - compare defect rates by programmer?
  - reward/punish managers?
- How do you compare class A to class B software, or orbiters to landers?
- Should contractor-written code be included in a center's metrics?
- Isn't a line of C worth more than a line of assembly code?

### Technical Issues

- How shall "lines" be counted?
  - Blank lines, comments, closing braces, macros, header files
- Should auto-generated code be counted?
- How should different software be classified?
  - Software vs. firmware
  - Flight vs. ground vs. test
  - Spacecraft vs. payload
  - ACS, Nav, C&DH, Instrument, science, uplink, downlink, etc
  - New, heritage, modified, COTS, GOTS

*Workshop Overview*

# NASA Fault Management Workshop

- **When:** April 13-15, 2008, New Orleans
- **Sponsor:** Jim Adams, Deputy Directory, Planetary Science
- **Web:** http://icpi.nasaprs.com/NASAFMWorkshop
- **Attendance:** ~100 people from NASA, Defense, Industry and Academia
- **Day 1:** Case studies + invited talk on history of spacecraft fault management.
  - "Missions of the future need to have their systems engineering deeply wrapped around fault management." (Gentry Lee, JPL)
- **Day 2:** Parallel sessions on (1) Architectures, (2) Verification & Validation, and (3) Practices/Processes/Tools + invited talk on importance of software architecture + poster session
  - "Fault management should be 'dyed into the design' rather than 'painted on'
  - "System analysis tools haven't kept pace with increasing mission complexity"
- **Day 3:** Invited talks on new directions in V&V and on model-based monitoring of complex systems + observations from attendees
  - "Better techniques for onboard fault management already exist and have been flown." (Prof. Brian Williams, MIT)

# What's Different About Flight Software?

## FSW has four distinguishing characteristics:

1. No direct user interfaces such as monitor and keyboard. All interactions are through uplink and downlink.
2. Interfaces with numerous flight hardware devices such as thrusters, reaction wheels, star trackers, motors, science instruments, temperature sensors, etc.
3. Executes on radiation-hardened processors and microcontrollers that are relatively slow and memory-limited. (Big source of incidental complexity)
4. Performs real-time processing. Must satisfy numerous timing constraints (timed commands, periodic deadlines, async event response). Being late = being wrong.

# Two Sources of Software Complexity

SW complexity = Essential complexity + Incidental complexi

- *Essential complexity* comes from problem domain and mission requirements
- Can reduce it only by descoping
- Can move it (e.g. to ops), but can't remove it

- *Incidental complexity* comes from choices about architecture, design, implementation, including avionics
- Can reduce it by making wise choices

# Good Description of Complexity

"*Complexity* is the label we give to the existence of many interdependent variables in a given system. The more variables and the greater their interdependence, the greater that system's complexity. Great complexity places high demands on a planner's capacities to gather information, integrate findings, and design effective actions. The links between the variables oblige us to attend to a great many features simultaneously, and that, concomitantly, makes it impossible for us to undertake only one action in a complex system. …

A system of variables is 'interrelated' if an action that affects or is meant to affect one part of the system will also affect other parts of it. Interrelatedness guarantees that an action aimed at one variable will have side effects and long-term repercussions."

# Factors that Increase Software Complexity

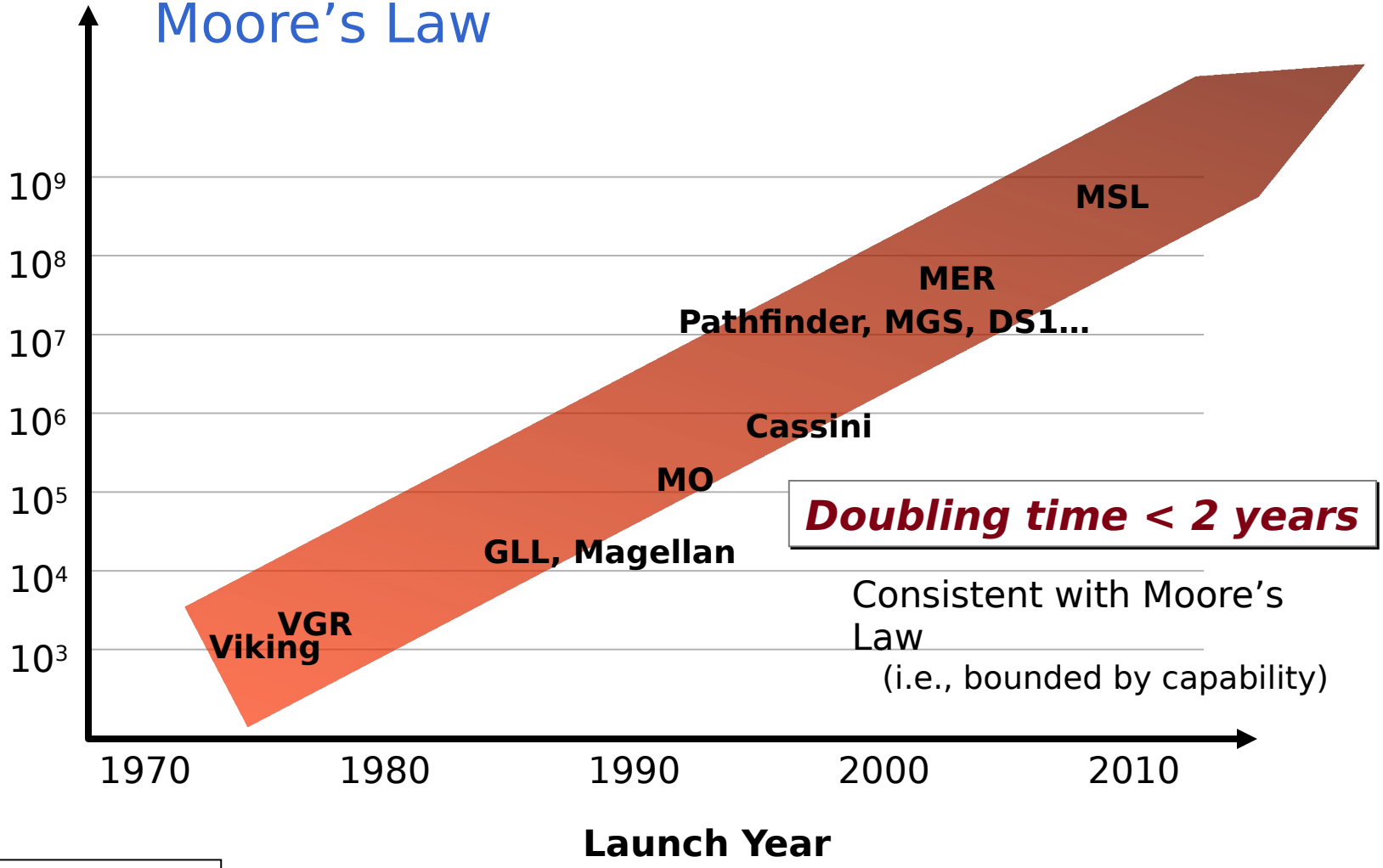
- **Human-rated Missions**
  - May require architecture redundancy and associated complexity
- **Fault Detection, Diagnostics, and Recovery (FDDR)**
  - FDDR requirements may result in complex logic and numerous potential paths of execution
- **Requirements to control/monitor increasing number of system components**
  - Greater computer processing, memory, and input/output capability enables control and monitor of more hardware components
- **Multi-threads of execution**
  - Virtually impossible to test every path and associated timing constraints
- **Increased security requirements**
  - Using commercial network protocols may introduce vulnerabilities
- **Including features that exceed requirements**
  - Commercial Off the Shelf (COTS) products or re-use code may provide capability that exceeds needs or may have complex interactions

Source: Cathy White, MSFC

# Flight Software Growth Trend: JPL Missions

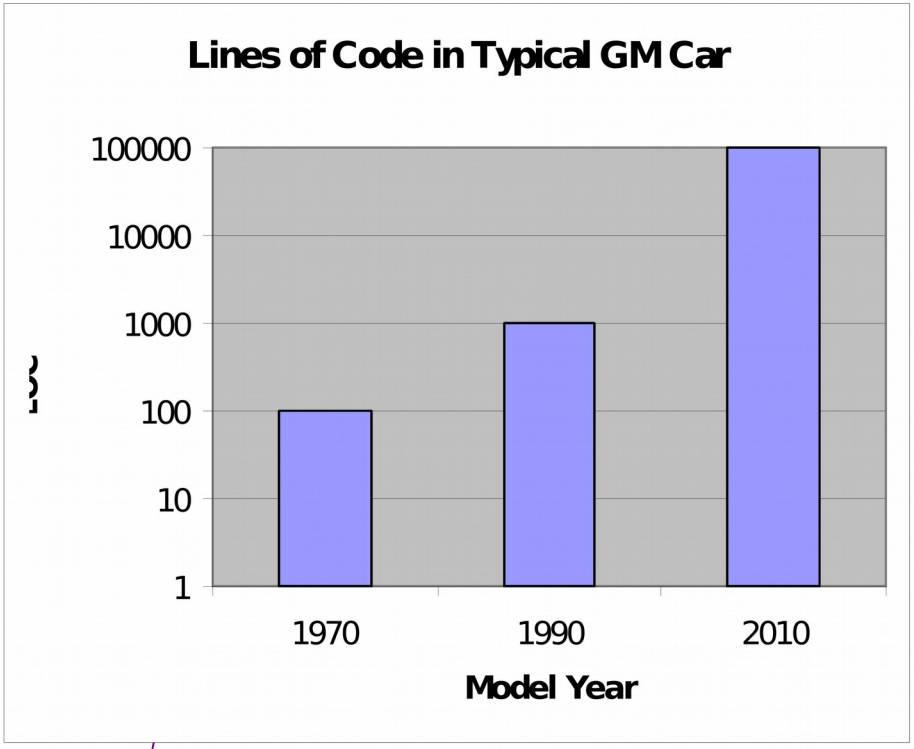
JPL data

With a vertical axis of size x speed, this chart shows growth keeping pace with Moore's Law




Source: Bob Rasmussen, JPL

# Growth in Automobile Software at GM




"Software per car will average 100 million lines of code by 2010 and is currently the single biggest expense in producing a car."

— Tony Scott, CTO, GM Information Systems & Services (2004)

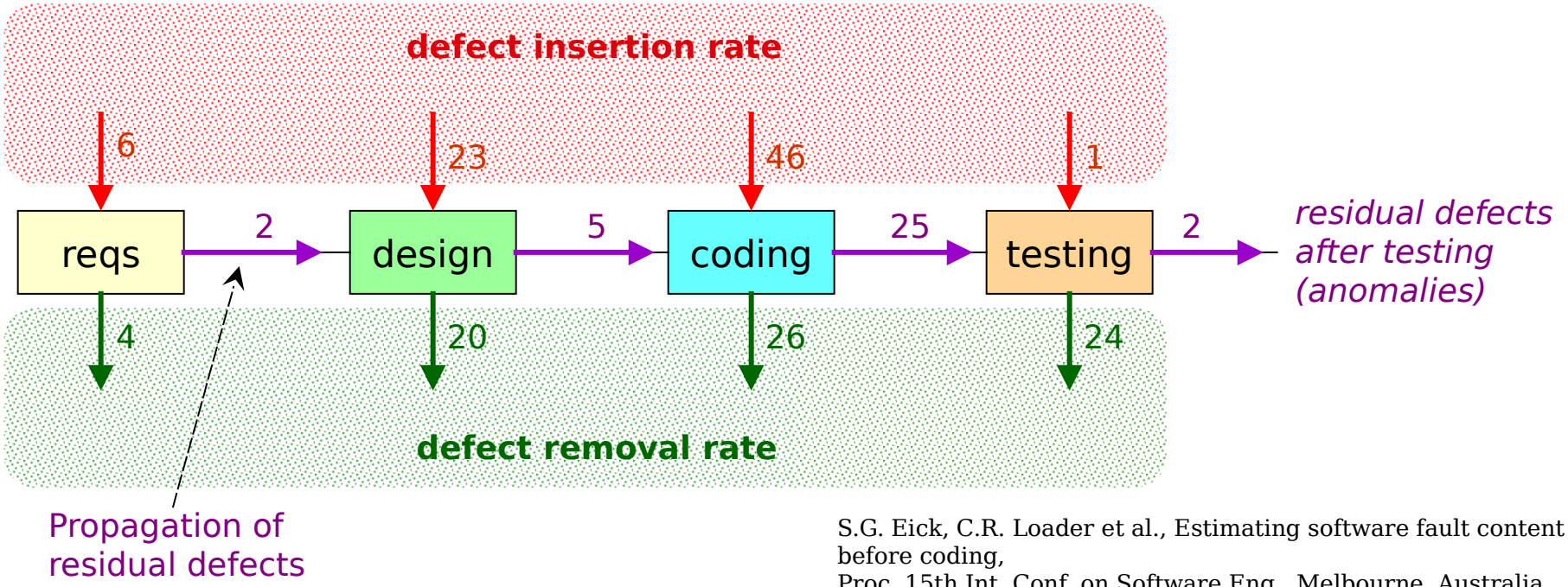
Note log scale!

References:

- www.techweb.com/wire/software/showArticle.jhtml?articleID=51000353
- www.eweek.com/c/a/Enterprise-Apps/GM-to-Software-Vendors-Cut-the-Complexity/

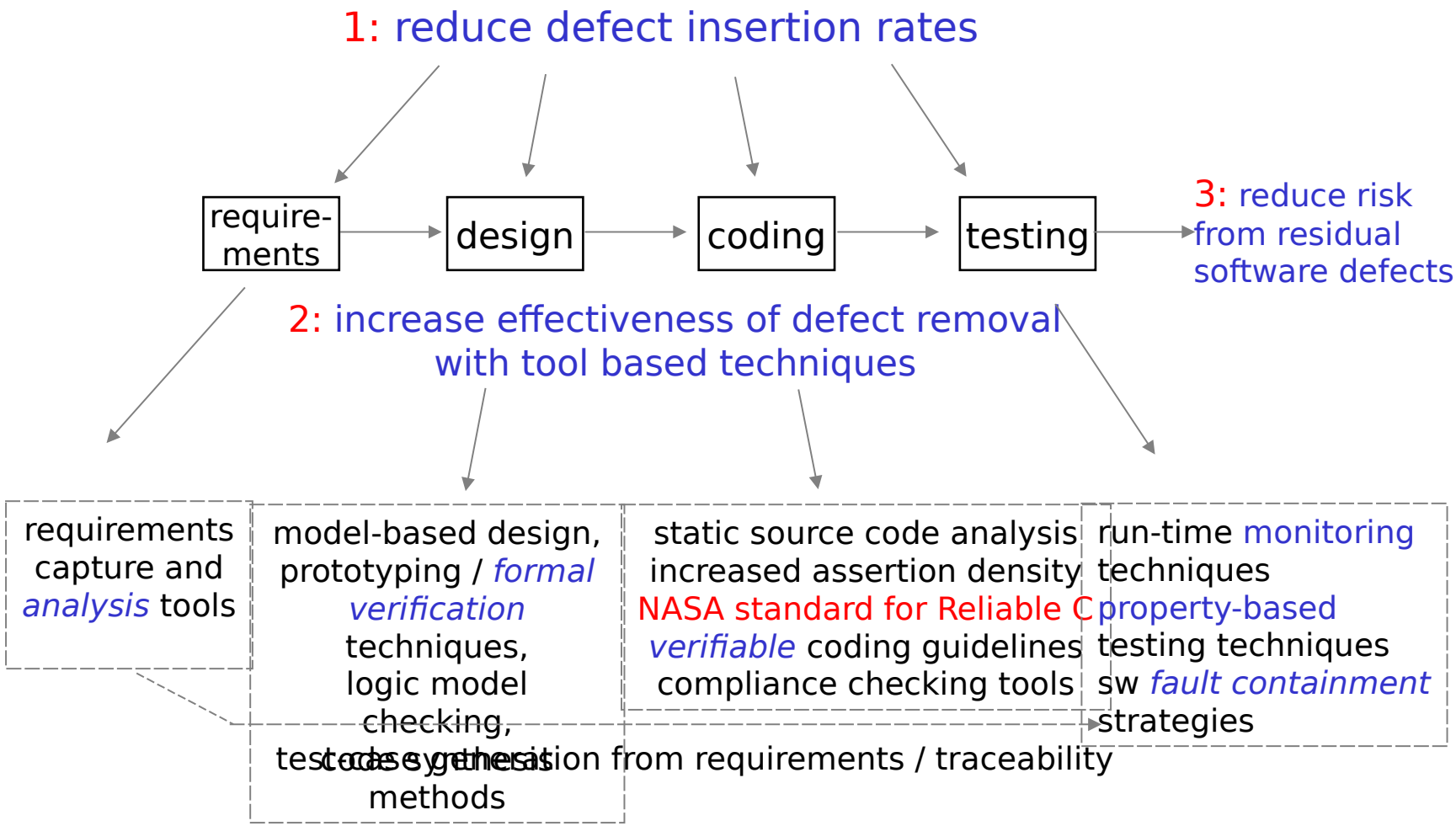
# Residual Defects in Software

- Each lifecycle phase involves human effort and therefore *inserts* some defects
- Each phase also has reviews and checks and therefore also *removes* defects
- Difference between the insertion and removal rates determines defect *propagation* rate
- the propagation rate at the far right determines the *residual defect rate*
- For a *good* industry-standard software process, residual defect rate is typically 1-10 per KNCSL
- For an *exceptionally good* process (e.g., Shuttle) it can be as low as 0.1 per KNCSL
- It is currently unrealistic to assume that it could be zero….




S.G. Eick, C.R. Loader et al., Estimating software fault content before coding,
Proc. 15th Int. Conf. on Software Eng., Melbourne, Australia, 1992, pp. 59-65

# Software Development Process
## for Safety- & Mission-Critical Code

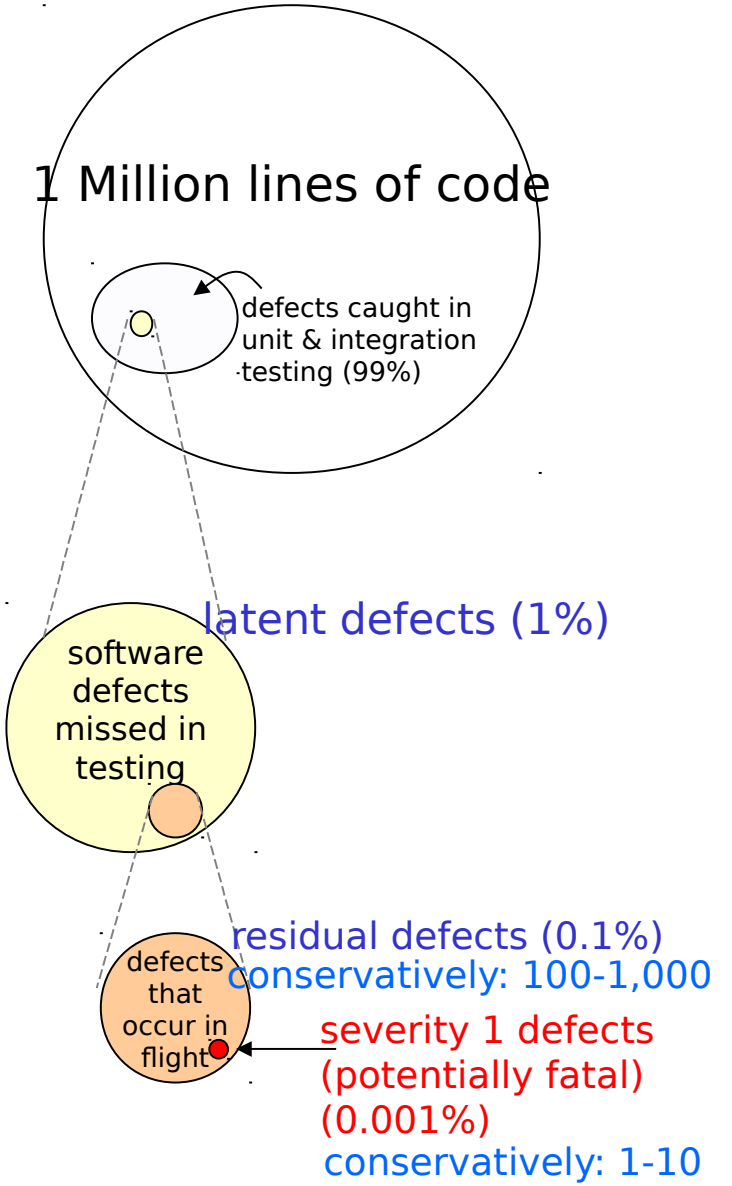

1: reduce defect insertion rates
require-ments
design
coding
testing
3: reduce risk from residual software defects
2: increase effectiveness of defect removal with tool based techniques
requirements capture and *analysis* tools
model-based design, prototyping / *formal verification* techniques, logic model checking, test case generation
code synthesis methods
static source code analysis
increased assertion density
NASA standard for Reliable C
*verifiable* coding guidelines
compliance checking tools
run-time monitoring techniques
property-based testing techniques
sw *fault containment* strategies
code generation from requirements / traceability

NASA/JPL Laboratory for Reliable Software

Source: Gerard Holzmann, JPL

# How good are state-of-the-art software testing methods?

- **Most estimates put the number of *residual defects* for a *good* software process at 1 to 10 per KNCSL**
  - A residual software defect is a defect missed in testing, that shows up in mission operations
  - A larger, but unknowable, class of defects is known as *latent software defects* – these are *all* defects present in the code after testing that *could* strike – only some of which reveal themselves as *residual defects* in a given interval of time.
- **Residual defects occur in any severity category**
  - A rule of thumb is to assume that the severity ratios drop off by powers of ten: if we use 3 severity categories with 3 being least and 1 most damaging, then 90% of the residual defects will be category 3, 9% category 2, and 1% category 1 (potentially fatal).
  - A mission with 1 Million lines of flight code, with a low residual defect ratio of 1 per KNCSL, then translates into 900 benign defects, 90 medium, and 9 potentially fatal residual software defects (i.e., these are defects that *will* happen, not those that *could* happen)

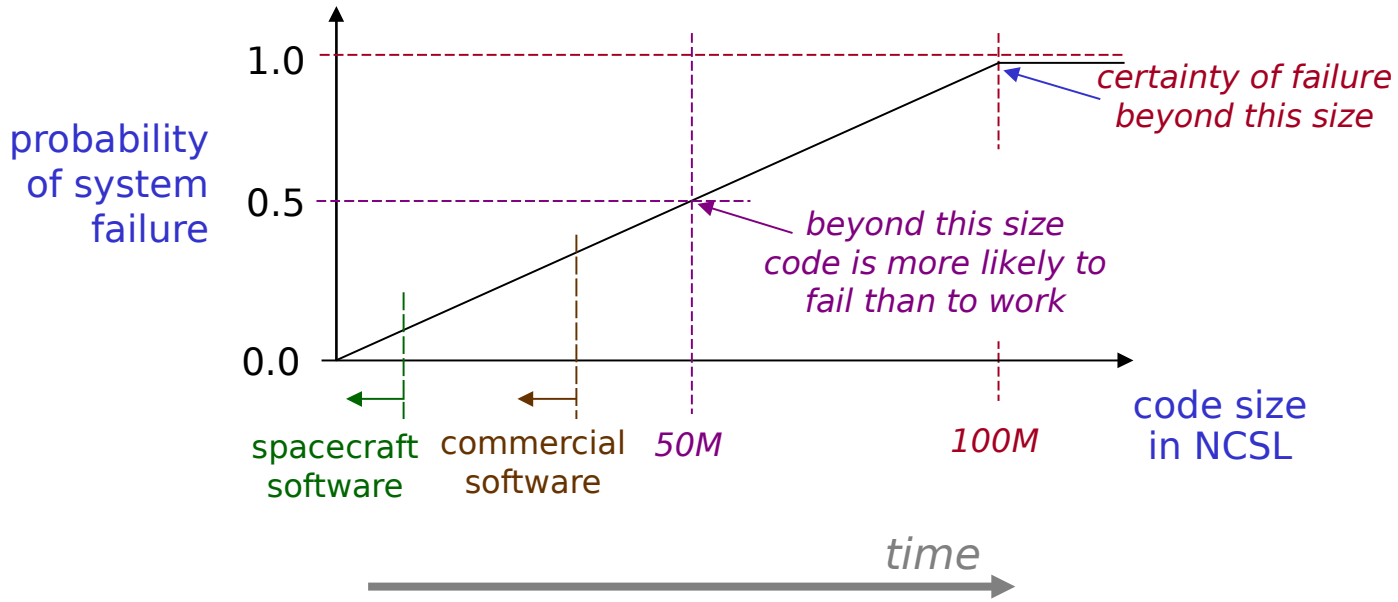

1 Million lines of code
defects caught in unit & integration testing (99%)
latent defects (1%)
software defects missed in testing
residual defects (0.1%)
conservatively: 100-1,000
defects that occur in flight
severity 1 defects (potentially fatal) (0.001%)
conservatively: 1-10

Source: Gerard Holzmann, JPL

# Is there a limit to software size?

**Assumptions:**

- 1 residual defect per 1,000 lines of code (industry average)
- 1 in every 100 residual defects occur in the 1st year of operation
- 1 in every 1000 residual defects can lead to mission failure
- System/software methods are at current state of the practice (2008)




*Long-term trend: increasing code size with each new mission*

# Impediments to Software Architecture within NASA

- Inappropriate modeling techniques
  - "Software architecture is just boxes and lines"
  - "Software architecture is just code modules"
  - "A layered diagram says it all"
- Misunderstanding about role of architectu in product lines and architectural reuse
  - "A product line is just a reuse library"
- Impoverished culture of architecture design
  - No standards for arch description and analysis
  - Architecture reviews are not productive
  - Architecture is limited to one or two phases
  - Lack of architecture education among engineers
- Failure to take architecture seriously
  - "We always do it that way. It's cheaper/easier/less risky to do it the way we did it last time."
  - "They do it a certain way 'out there' so we should too."
  - "We need to reengineer it from scratch because the mission is different from all others."

As presented by Prof. David Garlan (CMU) at NASA Planetary Spacecraft Fault Management Workshop, 4/15/08

## *Observations*

# Poor Software Practices within NASA

- No formal documentation of requirements
- Little to no user involvement during requirements definition
- Rushing to start design & code before requirements are understood.
- Wildly optimistic beliefs in re-use (especially when it comes to costing and planning).
- Planning to use new compilers, operating systems, languages, computers for the first time as if they were proven entities.
- Poor configuration management (CM)
- Inadequate ICDs
- User interfaces left up to software designers rather than prototyping and baselining as part of the requirements
- Big Bang Theory: All software from all developers comes together at end and miraculously works
- Planning that software will work with little or no errors found in every test phase.
- Poor integration planning (both SW-to-SW and SW-to-HW) (e.g., no early interface/integration testing)
- No pass/fail criteria at milestones (not that software is unique in this). Holding reviews when artifacts are not ready.
- Software too far down the program management hierarchy to have visibility into its progress
- Little to no life-cycle documentation
- Inadequate to no developmental metrics collected/analyzed
- No knowledgeable NASA oversight

An illustrative but incomplete list of poor software practices observed in NASA.

*John Hinkle, LaRC*

# NATO Software Engineering Conference 1968

- **This landmark conference, which introduced the term “software engineering”, was called to address “the software crisis”.**
- **Discussions of wide interest:**
  - problems of achieving sufficient reliability in software systems
  - difficulties of schedules and specifications on large software projects
  - education of software engineers

**Quotes from the 1968 report:**

“There is a widening gap between ambitions and achievements in software engineering.”

“Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death …”

“I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity?”

“The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.”

*Reference*

# Source Lines of Code

- Source lines of code (SLOC) is a software metric used to measure the size of a program by counting the number of lines in the program's source code.
- SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or effort once the software is produced.
- As a metric, SLOC dates back to line-oriented languages such as FORTRAN and assembler. In modern languages, one line of text does not necessarily correspond to a line of code.
- SLOC can be very effective at estimating *effort*, but less so at estimating *functionality*. It is not a good measure of productivity or of complexity-as-understandability.
- Data points:
  - Red Hat Linux 7.1 contains over 30 million lines of code
  - Boeing 777 has 4 million lines of code
  - GM: typical 1970 car had ~100 lines of code.
    By 1990, it was ~100K lines of code.
    By 2010, cars will average ~100 million lines of code.
    (Tony Scott, CTO, GM Information Systems & Services)

Measuring programming progress by lines of code is like measuring aircraft building progress by weight. —Bill Gates

# What is Architecture?

- Architecture is an essential systems engineering responsibility, which deals with the fundamental organization of a system, as embodied in its components and their relationships to each other and to the environment
  - Architecture addresses the structure, not only of the system, but also of its functions, the environment within which it will work, and the process by which it will be built and operated
- Just as importantly, however, architecture also deals with the principles guiding the design and evolution of a system
  - It is through the application and formal evaluation of architectural principles that complexity, uncertainty, and ambiguity in the design of complicated systems may be reduced to workable concepts
  - In the best practice of architecture, this aspect of architecture must not be understated or neglected

Source: Bob Rasmussen, JPL

# Some Essential Ideas

- Architecture is focused on fundamentals
  - An architecture that must regularly change as issues arise provides little guidance
  - Architecture and design are not the same thing
- Guidance isn't possible if the original concepts have little structural integrity to begin with
  - Choices must be grounded in essential need and solid principles
  - Otherwise, any migration away from the original high level design is easy to justify
- Even if the structural integrity is there, it can be lost if it is poorly communicated or poorly stewarded
  - The result is generally ever more inflexible and brittle

Source: Bob Rasmussen, JPL

# What is an Architect?

- An architect defines, documents, maintains, improves, and certifies proper implementation of an architecture — both its structure and the principles that guide it
  - An architect ensures through continual attention that the elements of a system come together in a coherent whole
  - Therefore, in meeting these obligations the role of architect is naturally concerned with leadership of the design effort throughout the development lifecycle
- An architect must ensure that…
  - The architecture (elements, relationships, principles) reflects fundamental, stable concepts
  - The architecture is capable of providing sound guidance throughout the whole process
  - The concept and principles of the architecture are never lost or compromised

Source: Bob Rasmussen, JPL

# Essential Activities

- Understand what a system must do
- Define a system concept that will accomplish this
- Render that concept in a form that allows the work to be shared
- Communicate the resulting architecture to others
- Ensure throughout development, implementation, and testing that the design follows the concepts and comes together as envisioned
- Refine ideas and carrying them forward to the next generation of systems

Source: Bob Rasmussen, JPL

# Architectural Activities in More Detail (1)

- Function
  - Help formulate the overall system objectives
  - Help stakeholders express what they care about in an actionable form
  - Capture in scenarios where and how the system will be used, and the nature of its targets and environment
  - Define the scope of the architecture, including external relationships
- Definition
  - Select and refine concepts on which the architecture might be based
  - Define essential properties concepts must satisfy, and the means by which they will be analyzed and demonstrated
  - Perform trades and assess options against essential properties — both to choose the best concept *and* to help refine objectives
- Articulation
  - Render selected concepts in elements that can be developed further
  - Choose carefully the structure and relationships among the elements
  - Identify the principles that will guide the evolution of the design
  - Express these ideas in requirements for the elements and their relationships that are complete, but preserve flexibility

Source: Bob Rasmussen, JPL

# Architectural Activities in More Detail (2)

- Communication
  - Choose how the architecture will be documented — what views need to be defined, what standards will be used to define them…
  - Create documentation of the architecture that is clear and complete, explaining all the choices and how implementation will be evaluated against high level objectives and stakeholder needs
- Oversight
  - Monitor the development, making corrections and clarifications, as necessary to the architecture, while enforcing it
  - Evaluate and test to ensure the result is as envisioned and that objectives are met, including during actual operation
- Advancement
  - Learn from others and document your experience and outcome for others to learn from
  - Stay abreast of new capabilities and methods that can improve the art

Source: Bob Rasmussen, JPL

# Software Architecture Reviews

Synopsis from “Architecture Reviews: Practice and Experience”, Maranzano et al, IEEE Software, March/April 2005.

- **Principles:**
  - A clearly defined problem statement drives the system architecture. Product line and business application projects require a system architect at all phases. Independent experts conduct reviews. Reviews are open processes. Conduct reviews for the *project’s* benefit.
- **Participants**
  - Project members, project management, review team (subject matter experts), architecture review board (a standing board)
- **Process**
  - 1: Screening. 2: Preparation. 3: Review meeting. 4: Follow-up.
- **Artifacts**
  - Architecture review checklist. Inputs (system requirements, functional requirements, architecture specification, informational documents). Outputs (set of issues, review report, optional management alert letter).
- **Benefits**
  - Cross-organizational learning is enhanced. Architecture reviews get management attention without personal retribution. Architecture reviews assist organizational change. Greater opportunities exist to find different defects in integration and system tests.

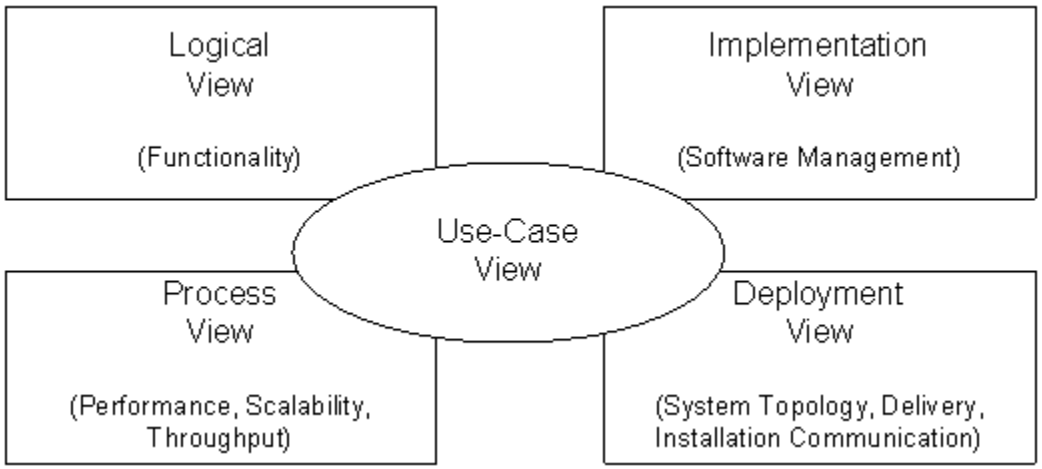
*Reference*

# What is Software Architecture?

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [11]
- Noteworthy points:
  - Architecture is an abstraction of a system that suppresses some details
  - Architecture is concerned with the *public interfaces* of elements and how they *interact* at runtime
  - Systems comprise more than one structure, e.g., runtime processes, synchronization relations, work breakdown, etc. No single structure is adequate.
  - Every software system has an architecture, whether or not documented, hence the importance of *architecture documentation*
  - The externally visible behavior of each element is part of the architecture, but not the internal implementation details
  - The definition is indifferent as to whether the architecture is good or bad, hence the importance of *architecture evaluation*

# What is a Reference Architecture?

- “A reference architecture is, in essence, a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.” [9]
- A reference architecture should be defined along different levels of abstraction, or “views”, thereby providing more flexibility in how it can be used. Ideally, these views map to the 4+1 Views of software architecture outlined in the Rational Unified Process and embodied in the RUP's Software Architecture Document.




*IBM Rational regards reference architecture as “the best of best practices”*

*Reference*

# What is Cyclomatic Complexity?

- *Cyclomatic complexity* measures path complexity
  - It counts the number of distinct paths through a method
- Various studies over the years have determined that methods having a cyclomatic complexity (or CC) greater than 10 have a higher risk of defects.
- Because CC represents the paths through a method, this is an excellent number for determining how many test cases will be required to reach 100 percent coverage of a method.

Source: *"In pursuit of code quality: Monitoring cyclomatic complexity"*, Andrew Glover, http://www.ibm.com/developerworks/java/library/j-cq03316/

# No Silver Bullet, But Reward the Stars

- In 1986 Fred Brooks wrote a widely-cited paper on software engineering “No Silver Bullet: essence and accidents of software engineering”
- The paper distinguished between essential complexity (from the problem to be solved) and accidental complexity (problems we create on our own, through our design and code).
- Brooks argues that:
  - No more technologies or practices that will serve as “silver bullets” and create a twofold improvement in programmer productivity over two years
  - Programming is a creative process: some designers are inherently better than others and are as much as 10-fold more productive
- Brooks advocates treating star designers equally well as star managers, providing them not just with equal remuneration, but also all the trappings of higher status (large office, staff, travel funds, etc.).

*Supports the recommendation to grow and promote software architects*

# Aerospace Corp. Software Activities

*Aerospace Corporation is engaged in the following activities to help address the software growth trend:*

- Re-invigorating software development standards
- Working to get systems engineering to properly flow down software reliability requirements
- Working with contractors to incorporate disciplined-rigorous software testing methodologies
- Educating the customer about software trends for space
- Hosting a space systems software reliability workshop
- Building an industry wide software development life cycle metrics database
- Recommending building and testing payload launch critical functionality first as an option
- Building models of the software development life cycle to proactively address software defect densities

*Douglas Buettner, "The Need for Advanced Space Software Development Technologies", Proceedings of the 23rd Aerospace Testing Seminar, Oct. 10-12, 2006*

# Flight Software Complexity Primer

*Table of Contents*


1. Science Requirements
2. Instrument and Sensor Accommodation
3. Inadequate Avionics Design
4. Hardware Interfaces and FSW Complexity
5. Miscellaneous Hardware Issues
6. Fear of Flight Software
7. Design for Testability
8. Effects of Flight Software on Mission Operations


This 10-page primer is included as an appendix in the report. Its purpose is to raise awareness of how seemingly reasonable decisions in one domain can have surprising consequences in another domain. The primer is an attempt to educate so that we can keep those surprises to a minimum.

# Audiences Briefed

| | |
|---|---|
| 4/22/08 | JPL Engineering Development Program Office |
| 5/01/08 | NASA Software Working Group (telecon) |
| 5/07/08 | NASA Engineering Management Board (Huntsville) |
| 6/12/08 | JPL Director Charles Elachi (Div. 31 technical visit) |
| 7/08/08 | JPL Interplanetary Network Directorate staff meeting |
| 7/10/08 | CMU Prof. David Garlan (expert on software architecture education) |
| 7/15/08 | Constellation Software & Avionics Control Panel (telecon) |
| 8/07/08 | NASA/Wallops Flight Facility system and software engineers |
| 8/20/08 | Missile Defense Agency (during their visit to JPL) |
| 9/15/08 | NASA Office of Chief Engineer (at HQ) |
| 9/16/08 | Office of Undersecretary of Defense, Software Engineering and System Assurance (DC) |
| 9/17/08 | NASA/Goddard Flight Software Branch |
| 9/17/08 | Applied Physics Lab, Johns Hopkins University |
| 9/26/08 | NASA Chief Engineer Mike Ryschkewitsch (telecon) |
| 9/29/08 | ESMD Software Risk Management Team (telecon) |
| 10/01/08 | NASA/Langley general audience |
| 10/06/08 | Pete Theisinger, Director, Engineering Systems Directorate |

# References


1. NATO Software Engineering Conference, 1968, Garmisch, Germany
2. Design, Development, Test and Evaluation (DDT&E) Considerations for Safe and Reliable Human-Rated Spacecraft Systems", NESC RP-06-108
3. ESMD Software Workshop (ExSoft 2007), April 10-12, 2007, Houston
4. NASA Planetary Spacecraft Fault Management Workshop, April 14-16, 2008, New Orleans
5. NPR 7150.2, NASA Software Engineering Requirements
6. NPR 7123.1A, NASA Systems Engineering Processes and Requirements
7. Product Requirements Development and Management Procedure, LMS-CP-5526 (NASA/LaRC)
8. Peer Review Inspection Checklists (a collection of checklists at LaRC, courtesy of Pat Schuler)
9. "Reference Architecture: The Best of Best Practices", http://www.ibm.com/developerworks/rational/library/2774.html
10. Gripe Session on Operations Complexity (JPL), 4/24/2008.
11. Software Architecture in Practice, 2nd edition, Bass, Clements, Kazman, 2003, Addison-Wesley.
12. "Architecture Reviews: Practice and Experience", Maranzano et al, IEEE Software, March/April 2005.